Journal of Discrete Algorithms 18 (2013) 22–31

Contents lists available at SciVerse ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Computing the longest common prefix array based on the Burrows–Wheeler transform

Timo Beller *, Simon Gog, Enno Ohlebusch, Thomas Schnattinger

Institute of Theoretical Computer Science, University of Ulm, 89069 Ulm, Germany

ARTICLE INFO

Article history:
Available online 4 August 2012

Keywords:
Longest common prefix array
Burrows–Wheeler transform
Wavelet tree
Shortest unique substrings
Shortest absent words

ABSTRACT

Many sequence analysis tasks can be accomplished with a suffix array, and several of them additionally need the longest common prefix array. In large scale applications, suffix arrays are being replaced with full-text indexes that are based on the Burrows–Wheeler transform. In this paper, we present the first algorithm that computes the longest common prefix array directly on the wavelet tree of the Burrows–Wheeler transformed string. It runs in linear time and a practical implementation requires approximately 2.2 bytes per character.

1. Introduction

A suffix tree for a string S of length n is a compact trie storing all the suffixes of S (so it is a full-text index). It is an extremely important data structure with applications in string matching, bioinformatics, and document retrieval, to mention only a few examples; see e.g. [14]. Suffix arrays can replace suffix trees and they use less memory than those. The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic ordering of the n suffixes of the string S. To be precise, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$, where S_i denotes the i-th suffix $S[i..n]$ of S; see Fig. 1 for an example. In the last decade, much effort has gone into the development of efficient suffix array construction algorithms (SACAs); see [38] for a survey.

However, the meteoric increase of DNA sequence information produced by next-generation sequencers demands new computer science approaches to data management because the data must be stored, analyzed, and mined. To analyze the massive quantities of data, established data structures like the suffix array (and the suffix tree) are being replaced by less space consuming data structures like the wavelet tree of the Burrows–Wheeler transformed sequence. It goes as follows: the sequence is subjected to the Burrows–Wheeler transform (BWT) [3], the Burrows–Wheeler transformed sequence is stored in a wavelet tree (or, more generally, in an FM-index [9]), and the wavelet tree [13] supports backward search on the original sequence. Let us recall the backward search technique in more detail. Let Σ be an ordered alphabet of size σ whose smallest element is the so-called sentinel character \$. In the following, S is a string (sequence) of length n over Σ having the sentinel character at the end (and nowhere else). The BWT transforms the string S into the string $\mathsf{BWT}[1..n]$ defined by $\mathsf{BWT}[i] = S[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $\mathsf{BWT}[i] = \$$ otherwise; see Fig. 1. Ferragina and Manzini [9] showed that it is possible to search a pattern backwards, character-by-character, in the suffix array SA of string S, without storing SA. Let $c \in \Sigma$ and ω be a substring of S. Given the ω-interval $[i..j]$ in the suffix array SA of S (i.e., ω is a prefix of $S_{\mathsf{SA}[k]}$ for all $i \leqslant k \leqslant j$, but ω is not a prefix of any other suffix of S), *backwardSearch*$(c, [i..j])$ returns the $c\omega$-interval

* Corresponding author.
E-mail addresses: Timo.Beller@uni-ulm.de (T. Beller), Simon.Gog@uni-ulm.de (S. Gog), Enno.Ohlebusch@uni-ulm.de (E. Ohlebusch), Thomas.Schnattinger@uni-ulm.de (T. Schnattinger).

1570-8667/\$ – see front matter
http://dx.doi.org/10.1016/j.jda.2012.07.007

i	SA	LCP	BWT	$S_{\mathsf{SA}[i]}$
1	19	-1	n	$
2	3	0	l	_anele_lepanelen$
3	9	1	e	_lepanelen$
4	4	0	_	anele_lepanelen$
5	13	5	p	anelen$
6	8	0	l	e_lepanelen$
7	1	1	$	el_anele_lepanelen$
8	6	2	n	ele_lepanelen$
9	15	3	n	elen$
10	17	1	l	en$
11	11	1	l	epanelen$
12	2	0	e	l_anele_lepanelen$
13	7	1	e	le_lepanelen$
14	16	2	e	len$
15	10	2	_	lepanelen$
16	18	0	e	n$
17	5	1	a	nele_lepanelen$
18	14	4	a	nelen$
19	12	0	e	panelen$
20		-1		

Fig. 1. Suffix array, LCP-array, and the Burrows–Wheeler-transformed string BWT of the string $S = \texttt{el\_anele\_lepanelen\$}$.

$[C[c] + Occ(c, i-1) + 1 \,..\, C[c] + Occ(c, j)]$, where $C[c]$ is the overall number of occurrences of characters in S which are strictly smaller than c, and $Occ(c, i)$ is the number of occurrences of the character c in $\mathsf{BWT}[1..i]$.

The approach described above was used for example in the software-tools Bowtie [24], BWA [26], SOAP2 [27], and 2BWT [23] for short read alignment (mapping short DNA sequences to a reference genome); see [10] for an overview article. More recently, it was suggested to use it also in *de novo* sequence assembly [41]. In the field of genome comparisons, this approach was first used by the software-tool *bbwt* [28], which uses k-mers (exact matches of fixed length k that appear in both sequences) as a basis of the comparison. It should be stressed that all these software-tools rely on the BWT (backward search) but not on the LCP-array. However, there is at least one algorithm that uses both BWT and LCP [35], and we expect that others will follow. The algorithm from [35] can be used in genome comparisons because it computes maximal exact matches (exact matches that cannot be extended in either direction towards the beginning or end without allowing for a mismatch) between two long strings (e.g. chromosomes).

In the last years, several algorithms have been proposed that construct the BWT either directly or by first constructing the suffix array and then deriving the BWT in linear time from it; see [29,18,42,36,8,4]. The latter approach has a major drawback: all known SACAs require at least $5n$ bytes of main memory provided that $n < 2^{32}$ (note that some real world data sets have a length that exceeds this bound; cf. Section 5). If one has to deal with large datasets, it is therefore advantageous to construct the BWT directly. For example, Okanohara and Sadakane [36] have shown that the SACA devised by Nong et al. [33] can be modified so that it directly constructs the BWT. Because it does not have to store the suffix array, it requires only $O(n \log \sigma \log\log_\sigma n)$ bits to construct the BWT [36] (ca. $2.5n$ bytes in practice; Sadakane, personal communication, 2011).

As described above, some sequence analysis tasks require the longest common prefix array (LCP-array): an array containing the length of the longest common prefix between every pair of consecutive suffixes in SA; see Fig. 1. Formally, the LCP-array is defined by $\mathsf{LCP}[1] = -1$, $\mathsf{LCP}[n+1] = -1$, and $\mathsf{LCP}[i] = |lcp(S_{\mathsf{SA}[i-1]}, S_{\mathsf{SA}[i]})|$ for $2 \leqslant i \leqslant n$, where $lcp(u, v)$ denotes the longest common prefix between the strings u and v. There are several linear time LCP-array construction algorithms (LACAs); see Section 2. They all first construct the suffix array and then obtain the LCP-array in linear time from it. So these LACAs suffer from the same drawback as mentioned above: at least $5n$ bytes of main memory are required. Here, we present the first LACA that acts directly on the Burrows–Wheeler transformed string and not on the suffix array.[1] The algorithm has a worst-case time complexity of $O(n \log \sigma)$. Hence its run-time is linear for a constant size alphabet. Moreover, we provide a practical implementation that requires approximately $2.2n$ bytes (throughout the paper, log stands for $\log_2$). Finally, we show that the approach can be used in other applications, namely finding shortest unique substrings and shortest absent words. A preliminary version of this article appeared in [1].

2. Related work

In their seminal paper [30], Manber and Myers did not only introduce the suffix array but also the longest-common-prefix array. They showed that both the suffix array and the LCP-array can be constructed in $O(n \log n)$ time for a string of

[1] The algorithms presented by Umar and Abdullah [44] do not really work on the BWT. They use the BWT to reconstruct data structures like the suffix array and then compute the LCP-array with the help of these data structures. Consequently, their algorithms are not space-efficient: they need at least $9n$ bytes.

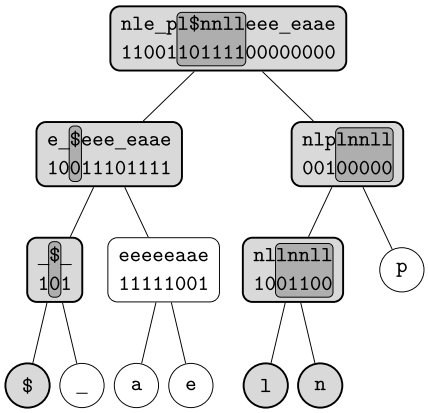

Fig. 2. Conceptual illustration of the wavelet tree of the string BWT = nle_pl$nnlleee_eaae. Only the bit vectors are stored; the corresponding strings are shown for clarity. The shaded regions will be explained later.

length n. Kasai et al. [22] gave the first linear time algorithm for the computation of the LCP-array. Their algorithm uses the string S, the suffix array, the inverse suffix array, and of course the LCP-array. Each of the arrays requires $4n$ bytes, thus the algorithm needs $13n$ bytes in total (for an ASCII alphabet). The main advantage of their algorithm is that it is simple and uses at most $2n$ character comparisons. But its poor locality behavior results in many cache misses, which is a severe disadvantage on current computer architectures. Manzini [31] reduced the space occupancy of Kasai et al.'s algorithm to $9n$ bytes with a slow down of about 5%–10%. He also proposed an even more space-efficient (but slower) algorithm that overwrites the suffix array. Recently, Kärkkäinen et al. [19] proposed another variant of Kasai et al.'s algorithm, which computes a permuted LCP-array (PLCP-array). In the PLCP-array, the lcp-values are in text order (position order) rather than in suffix array order (lexicographic order). This algorithm is much faster than Kasai et al.'s algorithm because it has a much better locality behavior. However, in virtually all applications lcp-values are required to be in suffix array order, so that in a final step the PLCP-array must be converted into the LCP-array. Although this final step suffers (again) from a poor locality behavior, the overall algorithm is still faster than Kasai et al.'s. Sirén [43] showed that the PLCP-array can be computed directly from a compressed suffix array. In a different approach, Puglisi and Turpin [39] tried to avoid cache misses by using the difference cover method of Kärkkäinen and Sanders [21]. The worst-case time complexity of their algorithm is $\mathcal{O}(nv)$ and the space requirement is $n+\mathcal{O}(n/\sqrt{v}+v)$ bytes, where v is the size of the difference cover. Experiments showed that the best run-time is achieved for $v = 64$, but their algorithm is still slower than Kasai et al.'s; see the experimental results in [19]. This is because it uses constant time range minimum queries, which take considerable time in practice. Just recently, Gog and Ohlebusch [12] presented a very space efficient and fast LACA, which trades character comparisons for cache misses.

3. Wavelet tree

The *wavelet tree* introduced by Grossi et al. [13] supports one backward search step in $O(\log\sigma)$ time. To explain this data structure, we may view the ordered alphabet Σ as an array of size σ so that the characters appear in ascending order in the array $\Sigma[1..\sigma]$, i.e., $\Sigma[1] = \$ < \Sigma[2] < \cdots < \Sigma[\sigma]$. We say that an interval $[l..r]$ is an *alphabet interval* if it is a subinterval of $[1..\sigma]$. For an alphabet interval $[l..r]$, the string $\mathsf{BWT}^{[l..r]}$ is obtained from the Burrows–Wheeler transformed string BWT of S by deleting all characters in BWT that do not belong to the sub-alphabet $\Sigma[l..r]$ of $\Sigma[1..\sigma]$. As an example, consider the string BWT = nle_pl$nnlleee_eaae and the alphabet interval $[1..4]$. The string $\mathsf{BWT}^{[1..4]}$ is obtained from nle_pl$nnlleee_eaae by deleting the characters l, n, and p. Thus, $\mathsf{BWT}^{[1..4]}$ = e_$eee_eaae. The wavelet tree of the string BWT over the alphabet $\Sigma[1..\sigma]$ is a balanced binary search tree defined as follows. Each node v of the tree corresponds to a string $\mathsf{BWT}^{[l..r]}$, where $[l..r]$ is an alphabet interval. The root of the tree corresponds to the string $\mathsf{BWT} = \mathsf{BWT}^{[1..\sigma]}$. If $l = r$, then v has no children. Otherwise, v has two children: its left child corresponds to the string $\mathsf{BWT}^{[l..m]}$ and its right child corresponds to the string $\mathsf{BWT}^{[m+1..r]}$, where $m = \lfloor\frac{l+r}{2}\rfloor$. In this case, v stores a bit vector $B^{[l..r]}$ whose i-th entry is 0 if the i-th character in $\mathsf{BWT}^{[l..r]}$ belongs to the sub-alphabet $\Sigma[l..m]$ and 1 if it belongs to the sub-alphabet $\Sigma[m+1..r]$. To put it differently, an entry in the bit vector is 0 if the corresponding character belongs to the left subtree and 1 if it belongs to the right subtree; see Fig. 2. Moreover, each bit vector B in the tree is preprocessed so that the queries $\mathit{rank}_0(B,i)$ and $\mathit{rank}_1(B,i)$ can be answered in constant time [17], where $\mathit{rank}_b(B,i)$ is the number of occurrences of bit b in $B[1..i]$. Obviously, the wavelet tree has height $O(\log\sigma)$. Because in an actual implementation it suffices to store only the bit vectors, the wavelet tree requires only $n\log\sigma$ bits of space plus $o(n\log\sigma)$ bits for the data structures that support rank queries in constant time.

Instead of reviewing the implementation of one backward search step on a wavelet tree, we present a generalization thereof: for an ω-interval $[i..j]$, the procedure *getIntervals*($[i..j]$) presented in Algorithm 1 returns the list of all $c\omega$-intervals. More precisely, it starts with the ω-interval $[i..j]$ at the root and traverses the wavelet tree in a depth-first manner as

Algorithm 1 For an ω-interval $[i..j]$, the function call *getIntervals*($[i..j]$) returns the list of all $c\omega$-intervals, and is defined as follows.

```
getIntervals([i..j])
  list ← [ ]
  getIntervals'([i..j], [1..σ], list)
  return list

getIntervals'([i..j], [l..r], list)
  if l = r then
    c ← Σ[l]
    add(list, [C[c] + i..C[c] + j])
  else
    (a0, b0) ← (rank0(B[l..r], i − 1), rank0(B[l..r], j))
    (a1, b1) ← (i − 1 − a0, j − b0)
    m ← ⌊(l+r)/2⌋
    if b0 > a0 then
      getIntervals'([a0 + 1..b0], [l..m], list)
    if b1 > a1 then
      getIntervals'([a1 + 1..b1], [m + 1..r], list)
```

follows. At the current node v, it uses constant time rank queries to obtain the number $b_0 - a_0$ of zeros in the bit vector of v within the current interval. If $b_0 > a_0$, then there are characters in $\mathsf{BWT}[i..j]$ that belong to the left subtree of v, and the algorithm proceeds recursively with the left child of v. Furthermore, if the number of ones is positive (i.e. if $b_1 > a_1$), then it proceeds with the right child in an analogous fashion. Clearly, if a leaf corresponding to character c is reached with current interval $[p..q]$, then $[C[c]+p\,..\,C[c]+q]$ is the $c\omega$ interval. In this way, Algorithm 1 computes the list of all $c\omega$-intervals. This takes $O(k \log \sigma)$ time for a k-element list. Because the wavelet tree has less than 2σ nodes, $O(\sigma)$ is another upper bound for the wavelet tree traversal. Consequently, Algorithm 1 has a worst-case time complexity of $O(\min\{\sigma, k \log \sigma\})$, where k is the number of elements in the output list. As an illustration of Algorithm 1, we compute all intervals of the form $c\mathtt{e}$ in the suffix array of Fig. 1 by invoking *getIntervals*([6..11]); note that [6..11] is the $\mathtt{e}$-interval. In the wavelet tree of Fig. 2, the visited nodes of the depth-first traversal are marked gray. The resulting list contains the three intervals [1..1], [13..15], and [17..18]. Algorithm 1 was developed by the fourth author [40] but others apparently had the same idea [6]; cf. also [11].

4. A LACA based on the BWT

Pseudo-code of our new LACA is given in Algorithm 2; it relies on Algorithm 1. We will illustrate the algorithm by an example. The first interval which is pulled from the queue is the ε-interval $\langle[1..19], 0\rangle$, and *getIntervals*([1..19]) returns a list of the seven ω-intervals [1..1], [2..3], [4..5], [6..11], [12..15], [16..18], and [19..19], where $|\omega| = 1$ for each ω. For every interval $[i_k..j_k]$, $\mathsf{LCP}[j_k + 1]$ is set to 0, except for the last one because $\mathsf{LCP}[20] = -1$. These six intervals are pushed on the queue, with the ℓ-value 1. Next, $\langle[1..1], 1\rangle$ is pulled from the queue. The list returned by *getIntervals*([1..1]) just contains the $\mathtt{n}\$$-interval [16..16]. Since $\mathsf{LCP}[17]$ has not been computed yet, $\langle[16..16], 2\rangle$ is pushed on the queue, $\mathsf{LCP}[17]$ is set to 1, and so on.

Algorithm 2 Computation of the LCP-array in $O(n \log \sigma)$ time.

```
initialize the array LCP[1..n + 1] /* i.e., LCP[i] = ⊥ for all 1 ⩽ i ⩽ n + 1 */
LCP[1] ← −1; LCP[n + 1] ← −1
initialize an empty queue
enqueue(⟨[1..n], 0⟩)
while queue is not empty do
  ⟨[i..j], ℓ⟩ ← dequeue()
  list ← getIntervals([i..j])
  for each [lb..rb] in list do
    if LCP[rb + 1] = ⊥ then
      enqueue(⟨[lb..rb], ℓ + 1⟩)
      LCP[rb + 1] ← ℓ
```

It is not difficult to see that Algorithm 2 maintains the following invariant: The set of the second components of all elements of the queue has either one element ℓ or two elements ℓ and $\ell + 1$, where $0 \leqslant \ell < n$. In the latter case, the elements with second component ℓ precede those with second component $\ell + 1$.

Theorem 4.1. *Algorithm 2 correctly computes the* LCP*-array.*

Proof. We proceed by induction on ℓ. In the base case, we have $\ell = 0$. For every character $c = \Sigma[k]$ occurring in S, the c-interval $[lb..rb]$ is in the list returned by *getIntervals*([1..n]), where $lb = C[c] + 1$ and $rb = C[d]$ with $d = \Sigma[k + 1]$. The algorithm sets $\mathsf{LCP}[rb + 1] = 0$ unless $rb = n$. This is certainly correct because the suffix $S_{\mathsf{SA}[rb]}$ starts with the character c

and the suffix $S_{\mathsf{SA}[rb+1]}$ starts with the character d. Clearly, because for every character c occurring in S the c-interval is in the list returned by *getIntervals*($[1..n]$), all entries of LCP with value 0 are set. Let $\ell > 0$. By the inductive hypothesis, we may assume that Algorithm 2 has correctly computed all lcp-values $< \ell$. After the last LCP-entry with value $\ell - 1$ has been set, the queue solely contains elements of the form $\langle [i..j], \ell \rangle$, where $[i..j]$ is the ω-interval of some substring ω of S with $|\omega| = \ell$. Let the $c\omega$-interval $[lb..rb]$ be in the list returned by *getIntervals*($[i..j]$). If $\mathsf{LCP}[rb+1] = \perp$, then we know from the induction hypothesis that $\mathsf{LCP}[rb+1] \geqslant \ell$, i.e., the suffixes $S_{\mathsf{SA}[rb]}$ and $S_{\mathsf{SA}[rb+1]}$ have a common prefix of length at least ℓ. On the other hand, $c\omega$ is a prefix of $S_{\mathsf{SA}[rb]}$ but not of $S_{\mathsf{SA}[rb+1]}$. Consequently, $\mathsf{LCP}[rb+1] < \ell + 1$. Altogether, we conclude that Algorithm 2 assigns the correct value ℓ to $\mathsf{LCP}[rb+1]$.

We still have to prove that all entries of the LCP-array with value ℓ are really set. So let k, $0 \leqslant k < n$, be an index with $\mathsf{LCP}[k+1] = \ell$. Since $\ell > 0$, the longest common prefix of $S_{\mathsf{SA}[k]}$ and $S_{\mathsf{SA}[k+1]}$ can be written as $c\omega$, where $c \in \Sigma$, $\omega \in \Sigma^*$, and $|\omega| = \ell - 1$. Consequently, ω is the longest common prefix of $S_{\mathsf{SA}[k]+1}$ and $S_{\mathsf{SA}[k+1]+1}$. Let $[i..j]$ be the ω-interval, p be the index with $\mathsf{SA}[p] = \mathsf{SA}[k] + 1$, and q be the index with $\mathsf{SA}[q] = \mathsf{SA}[k+1] + 1$. Clearly, $i \leqslant p < q \leqslant j$. Because ω is the longest common prefix of $S_{\mathsf{SA}[p]}$ and $S_{\mathsf{SA}[q]}$, there must be at least one index t with $p < t \leqslant q$ so that $\mathsf{LCP}[t] = |\omega| = \ell - 1$. If there is more than one index with that property, let t denote the smallest. According to the inductive hypothesis, Algorithm 2 assigns the value $\ell - 1$ to $\mathsf{LCP}[t]$. Just before that, a pair $\langle [s..t-1], \ell \rangle$ must have been pushed to the queue, where $[s..t-1]$ is some ω'-interval with $|\omega'| = \ell$. By the definition of t, we have $\mathsf{LCP}[r] > \ell - 1$ for all r with $p < r \leqslant t - 1$. Thus, p lies within the interval $[s..t-1]$. In other words, ω' is a prefix of $S_{\mathsf{SA}[p]}$. Moreover, $\mathsf{BWT}[p] = c$ implies that the $c\omega'$-interval, say $[lb..rb]$, is not empty. Since $\mathsf{BWT}[r] \neq c$ for all $p < r < q$, it follows that $rb = k$. At some point, $\langle [s..t-1], \ell \rangle$ is removed from the queue, and $[lb..k]$ is in the *list* returned by *getIntervals*($[s..t-1]$). Consequently, $\mathsf{LCP}[k+1]$ will be set to ℓ. □

Theorem 4.2. *Algorithm 2 has a worst-case time complexity of $O(n \log \sigma)$.*

Proof. In the proof, we use the ψ-function which is defined by $\psi(i) = \mathsf{ISA}[\mathsf{SA}[i]+1]$ for all i with $2 \leqslant i \leqslant n$ and $\psi(1) = \mathsf{ISA}[1]$, where ISA denotes the inverse of the permutation SA. Note that ψ is also a permutation of the suffix array.

We show that the algorithm creates at most $2n$ intervals. To this end, we first determine the maximum number of generated intervals $\langle [lb..rb], \ell \rangle$ for a fixed value rb. If $\ell < \mathsf{LCP}[rb+1]$, then there exists no such interval because otherwise the value $\mathsf{LCP}[rb+1]$ would be set to ℓ. Moreover, for any $\ell > \mathsf{LCP}[\psi[rb]+1] + 1$ the algorithm does not create an interval which ends at position rb. This can be seen as follows. There is only one interval $\langle [x..y], \ell - 1 \rangle$ which can generate the interval of length ℓ ending at rb. Clearly, this interval must contain $\psi[rb]$. So $x \leqslant \psi[rb] \leqslant y$. It follows from $\ell - 1 > \mathsf{LCP}[\psi[rb]+1]$ that $y = \psi[rb]$. If this interval would be added to the *queue*, then $\mathsf{LCP}[\psi[rb]+1]$ would be set to $\ell - 1$. However, this is impossible.

To sum up, all created intervals $\langle [lb..rb], \ell \rangle$ for a fixed value of rb satisfy $\mathsf{LCP}[rb+1] \leqslant \ell \leqslant \mathsf{LCP}[\psi[rb]+1]+1$. Therefore, the algorithm generates at most $\mathsf{LCP}[\psi[rb]+1] - \mathsf{LCP}[rb+1] + 2$ intervals ending at rb. (Note that this number is always positive because $\mathsf{LCP}[\psi[rb]+1] \geqslant \mathsf{LCP}[rb+1] - 1$.) Summing over all possible end positions rb gives an upper bound on the number of created intervals:

$$\sum_{rb=1}^{n} \big(\mathsf{LCP}\big[\psi[rb]+1\big] - \mathsf{LCP}[rb+1] + 2\big) = 2n + \sum_{rb=1}^{n} \mathsf{LCP}\big[\psi[rb]+1\big] - \sum_{rb=1}^{n} \mathsf{LCP}[rb+1] = 2n$$

The function *getIntervals* takes $O(\log \sigma)$ time for each interval, whereas the remaining statements of Algorithm 2 take constant time per interval. Thus, the run-time of Algorithm 2 is in $O(n \log \sigma)$. □

Algorithm 2 uses only the wavelet tree of the BWT of S, a queue to store the ω-intervals and the LCP-array. Because a practical implementation should use as little space as possible, we next show how to reduce the space consumption of the latter two. The second components (ℓ-values) of the queue entries need not be stored because one can simply count how many ω-intervals with $|\omega| = \ell$ enter the queue; see Algorithm 3 for details. For a fixed ℓ, the ω-intervals with $|\omega| = \ell$ do not overlap. Thus, they can be stored in two bit vectors, say B and E of size n, and an ω-interval $[i..j]$ is stored by setting the bits $B[i]$ and $E[j]$ (due to singleton intervals, we actually need two bit vectors). By the invariant mentioned above, at any point in time Algorithm 2 has to deal with at most two different ℓ-values. Therefore, we can replace the queue with four bit vectors of length n. The price to be paid for this is an increase in the worst-case time complexity. For each ℓ-value, two bit vectors of length n are scanned to determine all ω-intervals with $|\omega| = \ell$. So the number of scans is proportional to the maximum lcp-value. Since this can be $n - 2$ (consider the string $S = a^{n-1}\$$), the time complexity rises to $O(n^2)$. (However, in practice the maximum lcp-value is only a fraction of n [25], and on average it is $O(\log n)$ [7].)

For this reason, we prefer the following hybrid approach. For a fixed ℓ, if there are more than $\frac{n}{2 \log n}$ ω-intervals with $|\omega| = \ell$ (note that this can happen only $O(\log n)$ times), we use the bit vectors; otherwise we use the queue. More precisely, we start with the queue, switch to the bit vectors when there are more than $\frac{n}{2 \log n}$ ω-intervals with the current ℓ-value, and switch back if there are less than $\frac{n}{2 \log n}$ ω-intervals with the current ℓ-value. Note that the queue uses at most n bits because each queue entry is an interval that can be represented by two numbers using $\log n$ bits each. This hybrid approach does not increase the worst-case time complexity as we show next. There are $\frac{n}{\log n}$ blocks of length $\log n$ in a bit vector of

length n. Since the two bit vectors have to be scanned at most $\log n$ times, we must handle at most $2n$ blocks. In each block we determine the leftmost 1-bit. If we find such a 1-bit, we unset this bit and search again for the leftmost 1-bit in the same block. Because every interval creates only two 1-bits, this happens at most $4n$ times (remember that there are at most $2n$ intervals). Altogether, we get all stored intervals by searching at most $6n$ times the leftmost 1-bit in a block of $\log n$ bits. This search can be done in constant time under the word-RAM model with word size $\Omega(\log n)$, thus the intervals can be managed in bit vectors with $O(n)$ time complexity. Our experiments showed that this hybrid implementation of Algorithm 2 is actually faster than a implementation which solely uses the queue (from the STL of C++). This can be attributed to fewer cache misses due to the fact that the bit vectors are accessed sequentially.

Up to now, our LACA needs $n \log \sigma$ bits for the wavelet tree of the BWT plus $o(n \log \sigma)$ bits for the data structures that support rank queries in constant time, $4n$ bits for the storage of the ω-intervals, and $4n$ bytes for the LCP-array itself. Our goal is to stay below $2.5n$ bytes because this is (currently) the space that is needed to build the BWT; cf. Section 1. To meet this goal, we stream the LCP-array to disk. This is possible because Algorithm 2 calculates lcp-values in ascending order. Clearly, the LCP-array requires only $k \cdot n$ bits to store all lcp-values less than 2^k. During the computation of these lcp-values, the i-th bit of a bit vector D of length n is set when a value is assigned to LCP[i]. Afterwards the LCP-array is written to disk, but the bit vector D tells us which LCP-entries are already done and we preprocess D so that rank queries can be answered in constant time. Let m be the number of zeros in D. We use a new array A of length m that also occupies $k \cdot n$ bits. In other words, each array element of A consists of $b = \lfloor \frac{k \cdot n}{m} \rfloor$ bits, which are initially set to zero. Then, we compute all lcp-values less than $2^k + 2^b - 1$. When a value ℓ is to be assigned to LCP[i], we store the value $\ell - 2^k + 1$ in $A[rank_0(D, i)]$. After all lcp-values less than $2^k + 2^b - 1$ have been computed, we further proceed as follows. During a scan of the bit vector D, we count the number of zeros seen so far. So when an index i with $D[i] = 0$ is encountered, we know that this is, say, the j-th zero seen so far. Now we use a case analysis. If $A[j] = 0$, then LCP[i] has not been computed yet and there is nothing to do. Otherwise, the value $2^k - 1 + A[j]$ is written at index i to the LCP-array on the disk, and $D[i]$ is set to one. When the scan of D is completed, the (updated) bit vector D is preprocessed so that rank queries can be answered in constant time. This process is iterated (calculate the new values of m and b, initialize a new array A, etc.) until the LCP-array is completely filled.

For example, for DNA sequences the wavelet tree takes $2n$ bits and the rank data structures take less than $2n$ bits. Furthermore, the storage of the ω-intervals takes $4n$ bits and the bit vector D plus its rank data structure take $1.25n$ bits. Because we want to use at most $20n$ bits, $k = 10$ is the right choice for DNA sequences.

We can save even more space by splitting our algorithm in two phases. In the first phase, we do not keep the LCP-array in main memory. Instead, at each step the current lcp-value and the indices at which this value appears in the LCP-array are written sequentially into a file. This phase needs only the wavelet tree, $4n$ bits for storing the intervals and n bits for the D vector. In the second phase, we read lcp-values together with their indices from file and fill the LCP-array. This phase needs the same memory as phase 1 provided that the parameter k is chosen appropriately.

In several applications, the access to the LCP-array is sequential, so it can be streamed from disk. If random access is needed, one can get a compressed representation of the LCP-array by streaming it from disk, and then this compressed version is kept in main memory. There are several compressed versions of the LCP-array which use about 1 byte per entry in practice, while the access time remains essentially the same as for the uncompressed version; see e.g. [2]. Moreover, the tree topology of a compressed suffix tree can be constructed by streaming the LCP-array from disk and using a succinct stack which uses only $2n$ bits of space; see [34].

A reviewer of the preliminary version of this article [1] came up with the following idea to improve the practical runtime for finding the next 1-bit in the bit vector: "Instead of one bit vector (let us call it B_0) one can keep a sequence of bit vectors B_0, $B_1, \ldots$ The invariant of such a data structure is: $B_{i+1}[j] = 0 \Leftrightarrow B_i[8j] = B_i[8j+1] = \cdots = B_i[8j+7] = 0$. One can view it as a bit-encoded balanced 8-ary tree. Using an additional array of constant size (256 bytes), the first bit set can be identified in $\log_8 n$ operations, and when reset to 0, the data-structure can be updated in at most $\log_8 n$ another operations." We implemented this method with a 64-ary tree, but experiments showed that it is not faster than ours (probably because the bit vector is not sparse enough).

5. Experimental results

All programs were compiled using the `gcc` version 4.4.3 with the options `-O9 -msse4.2 -DNDEBUG` on a 64 bit Ubuntu (Kernel 2.6.32) system equipped with a six-core AMD Opteron processor 2431 with 2.4 GHz and 32 GB of RAM. The data originates from the Pizza&Chili Corpus (http://pizzachili.dcc.uchile.cl/). For space reasons, we report only the results for files of 200 MByte. Additionally, the genome of the house mouse (NCBIm36, http://www.ensembl.org/Mus_musculus) was used. We compared our new algorithm and the space efficient version of it (called new algorithm 2 in Table 1) with the KLAAP-algorithm of Kasai et al. [22], the Φ and Φ64-semi[2] algorithms of Kärkkäinen et al. [19], and the go-Φ algorithm of Gog and Ohlebusch [12]. The suffix array construction was done by Mori's libdivsufsort-algorithm[3] (http://code.google.com/p/libdivsufsort/), while the direct BWT construction is due to Okanohara and Sadakane [36]. The

[2] The semi-external version of Φ64 is one of the fastest algorithms of the Φ family.

[3] Because the 32 bit version is limited to files of size $\leqslant 2^{31}$, we had to use the 64 bit version for the mouse genome (which needs $9n$ bytes).

Table 1
Experimental results: for each file, the first column shows the real run-time in seconds and the second column shows the maximum memory usage per character. As an example, consider the dna 200 MB file. The construction of its suffix array takes 65 s and 5n bytes (1000 MB), whereas the direct construction of its BWT takes 85 s and 1.9n bytes (380 MB). Rows 3–8 refer to the construction of the LCP-array under the assumption that the suffix array (the BWT, respectively) has already been built. In rows 9–14, the first column shows the overall run-time and the second shows the overall maximum memory usage per character. For a fair comparison of the run-time, the data was chosen in such a way that all data structures fit in the main memory. Of course, for very large files the space usage of 9n bytes is disadvantageous because then the data structures must reside in secondary memory, and this slows down the algorithms.

	dna 200 MB		english 200 MB		proteins 200 MB		sources 200 MB		xml 200 MB		mouse 3242 MB	
SA constr.	65	5.0	64	5.0	71	5.0	43	5.0	47	5.0	1689	8.9
BWT constr.	85	1.9	106	2.2	139	2.6	82	2.1	82	2.1	1480	2.0
KLAAP	48	9.0	46	9.0	45	9.0	28	9.0	29	9.0	1501	9.0
Φ	34	9.0	29	9.0	28	9.0	21	9.0	21	9.0	1117	9.0
Φ64-semi	74	1.0	67	1.0	67	1.0	53	1.0	68	1.0	1509	1.0
go-Φ	48	2.0	69	2.0	66	2.0	48	2.0	45	2.0	1297	2.2
new algorithm	45	1.9	105	2.2	114	2.3	94	2.2	66	2.2	814	2.0
new algorithm 2	53	1.0	110	1.3	116	1.3	100	1.5	72	1.4	962	1.0
KLAAP	114	9.0	110	9.0	116	9.0	71	9.0	77	9.0	3191	9.0
Φ	99	9.0	93	9.0	99	9.0	64	9.0	68	9.0	2806	9.0
Φ64-semi	140	5.0	131	5.0	138	5.0	97	5.0	115	5.0	3198	8.9
go-Φ	114	5.0	133	5.0	137	5.0	91	5.0	93	5.0	2986	8.9
new algorithm	131	1.9	211	2.2	253	2.6	176	2.2	149	2.2	2294	2.0
new algorithm 2	139	1.9	216	2.2	255	2.6	183	2.1	154	2.1	2442	2.0

KLAAP-algorithm is our own implementation, all other programs were kindly provided by the authors. Looking at the experimental results in Table 1, one can see that the Φ algorithm is the fastest LACA. However, in large scale applications its space usage of 9n bytes is the limiting factor. The memory usage of algorithm go-Φ (row 6) is similar to that of our algorithm but it relies on the suffix array, so its overall space usage (row 12) is due to the suffix array construction (row 1). By contrast, our new algorithm solely depends on the BWT, so that its overall maximum memory usage per character is approximately 2.2n bytes (row 13). It can be attributed to the usual space–time trade-off that our new algorithm is the slowest LACA in the contest for the 200 MB files (except for small alphabets). However, this changes in large scale applications when memory is tight.

6. Other applications

In this section, we will sketch two other applications of our new approach. Because \$ is solely used to mark the end of the string S, we have to exclude it in the considerations below.

6.1. Shortest unique substrings

As a first application, we will briefly describe how to find shortest unique substrings. This is relevant in the design of primers for DNA sequences; see [14].

Definition 6.1. A substring $S[i..j]$ is *unique* if it occurs exactly once in S. The *shortest unique substring problem* is to find all shortest unique substrings of S.

Clearly, every suffix of S is unique because S is terminated by the special symbol \$. Since we are not interested in these, we will exclude them. One can show that the $(\ell+1)$-length prefix of $S_{\mathsf{SA}[i]}$, where $\ell = \max\{\mathsf{LCP}[i], \mathsf{LCP}[i+1]\}$, is the shortest unique substring of S that starts at position $\mathsf{SA}[i]$. Using this observation, we can modify Algorithm 2 so that it computes a shortest unique substring of S in $O(n \log \sigma)$ time. The resulting Algorithm 3 can easily be changed so that it computes all shortest unique substrings or even all unique substrings of S. We make use of the fact that Algorithm 2 computes lcp-values in ascending order. So when Algorithm 2 executes the statement $\mathsf{LCP}[rb+1] \leftarrow \ell$ and $\mathsf{LCP}[rb]$ has been set before, then $\max\{\mathsf{LCP}[rb], \mathsf{LCP}[rb+1]\} = \ell$ and $S[\mathsf{SA}[rb]..\mathsf{SA}[rb]+\ell]$ is the shortest unique substring of S that starts at position $\mathsf{SA}[rb]$. Analogously, if $\mathsf{LCP}[rb+2]$ has been set before, then $\max\{\mathsf{LCP}[rb+1], \mathsf{LCP}[rb+2]\} = \ell$ and $S[\mathsf{SA}[rb+1]..\mathsf{SA}[rb+1]+\ell]$ is the shortest unique substring of S that starts at position $\mathsf{SA}[rb+1]$. Because the current value of ℓ is always available, all we have to know is whether or not $\mathsf{LCP}[rb]$ ($\mathsf{LCP}[rb+2]$, respectively) has been computed before. Consequently, we can replace the LCP-array with the bit vector D of length n, and $D[i]$ is set to one instead of assigning a value to $\mathsf{LCP}[i]$. However, there are two subtleties that need to be taken into account. First, the suffix array is not at hand, so we have to find an alternative way to output the string $S[\mathsf{SA}[rb]..\mathsf{SA}[rb]+\ell]$. Second, we have to exclude this string if it is a suffix. Fortunately, the wavelet tree provides the needed functionality, as we shall see next. The LF-mapping is defined by $LF(i) = \mathsf{ISA}[\mathsf{SA}[i]-1]$ for all i with $\mathsf{SA}[i] \neq 0$ and $LF(i) = 0$ otherwise (where ISA denotes the inverse of the permutation SA). Its long name *last-to-first column mapping* stems from the fact that it maps the last column

$L = \mathsf{BWT}$ to the first column F, where F contains the first character of the suffixes in the suffix array, i.e., $F[i] = S[\mathsf{SA}[i]]$. More precisely, if $\mathsf{BWT}[i] = c$ is the k-th occurrence of character c in BWT, then $j = LF(i)$ is the index such that $F[j]$ is the k-th occurrence of c in F. We recall that the ψ-function, defined by $\psi(i) = \mathsf{ISA}[\mathsf{SA}[i] + 1]$ for all i with $2 \leqslant i \leqslant n$ and $\psi(1) = \mathsf{ISA}[1]$, is the inverse permutation of LF. With the wavelet tree, both $LF(i)$ and $\psi(i)$ can be computed in $O(\log\sigma)$ time; see [32]. Moreover, the character $F[i]$ can be determined in $O(\log\sigma)$ time by a binary search on C,[4] or, using $n + o(n)$ extra bits, by a constant time rank query. Since $S[\mathsf{SA}[rb]] = F[rb]$, $S[\mathsf{SA}[rb] + 1] = F[\psi(rb)]$, $S[\mathsf{SA}[rb] + 2] = F[\psi(\psi(rb))]$ etc., it follows that the string $S[\mathsf{SA}[rb]..\mathsf{SA}[rb] + \ell]$ coincides with $F[rb]\,F[\psi(rb)]\ldots F[\psi^{\ell}(rb)]$ (which can be computed in $O(\ell\log\sigma)$ time). This solves our first little problem. The second problem was to exclude suffixes from the output. This can be done by keeping track of the suffix of length $\ell + 1$, where ℓ is the current length. To be precise, initially $\ell = 0$ and the suffix of length 1 is the character \$, which appears at index $idx = 1$. Every time ℓ is incremented, we obtain the index of the suffix of length $\ell + 1$ by the assignment $idx \leftarrow LF(idx)$. Consequently, a unique substring at index rb is output only if $rb \neq idx$.

Algorithm 3 Computation of a shortest unique substring.

```
initialize a bit vector D[1..n+1] /* i.e., D[i] = 0 for all 1 ⩽ i ⩽ n+1 */
D[1] ← 1; D[n+1] ← 1
initialize an empty queue
enqueue([1..n])
ℓ ← 0; size ← 1; idx ← 1
while queue is not empty do
  if size = 0 then
    ℓ ← ℓ + 1
    size ← current size of the queue
    idx ← LF(idx)
  [i..j] ← dequeue()
  size ← size − 1
  list ← getIntervals([i..j])
  for each [lb..rb] in list do
    if D[rb + 1] = 0 then
      enqueue([lb..rb])
      D[rb + 1] ← 1
      if D[rb] = 1 and rb ≠ idx then
        return F[rb] F[ψ(rb)] ... F[ψ^ℓ(rb)] /* string S[SA[rb]..SA[rb] + ℓ] */
      if D[rb + 2] = 1 and rb + 1 ≠ idx then
        return F[rb + 1] F[ψ(rb + 1)] ... F[ψ^ℓ(rb + 1)]
```

6.2. Shortest absent words

As a second application, we show that our approach allows us to compute shortest absent words. This is relevant because short DNA sequences that do not occur in a genome are interesting to biologists. For example, the fact that the human genome does not contain all possible DNA sequences of length 11 may be due to negative selection. For this reason, several algorithms have been developed that compute shortest absent words; see [5,15,16,37,45]. Our approach allows us to give an alternative algorithm for this task.

Definition 6.2. Given string S, a string $\omega \in (\Sigma \setminus \{\$\})^{+}$ is called an *absent word* if it is not a substring of S. The *shortest absent words problem* is to find all shortest absent words.

In the following, let σ be the size of $\Sigma \setminus \{\$\}$. If $\sigma = 1$, say $\Sigma \setminus \{\$\} = \{a\}$, then S consists solely of a's. That is, $S = a^n$, where n is the length of S. In this case, the shortest absent word is obviously a^{n+1}. Therefore, we may assume that $\sigma > 1$. We define a tree structure as follows. The root of the interval-tree is the ε-interval $[1..n]$, and the children of an ω-interval $[i..j]$ are the $c\omega$-intervals returned by *getIntervals*$([i..j])$, where $c \neq \$$. Clearly, if the ω-interval $[i..j]$ has less than σ children, then every non-occurring child corresponds to an absent word. In other words, if for some $c \in \Sigma \setminus \{\$\}$, the $c\omega$-interval is empty, then $c\omega$ is not a substring of S. Our algorithm can be viewed as a depth-first traversal through this (virtual) tree. During the traversal, the shortest absent words that have been detected so far are stored in a list, while a variable *len* stores their length. For each absent word ω with $|\omega| \leqslant len$, we proceed by case distinction. If $|\omega| = len$, then ω is added to the list. Otherwise, if $|\omega| < len$, then *len* is set to $|\omega|$ and the list is set to $[\omega]$. We next show that the initial value of *len* can be upper bounded without sacrificing correctness. Since there are at most $n - k$ substrings of length k in S, while there are σ^k possible strings of length k, it follows that there must be an absent word of length k provided that $\sigma^k > n - k$. In particular, there must be an absent word of length $\lceil \log_\sigma n \rceil$. Therefore, if we initially set *len* to $\lceil \log_\sigma n \rceil$, the algorithm will find all shortest absent words. For how many substrings ω of S must the procedure *getIntervals* be applied? Clearly, it need not be

[4] As pointed out by an anonymous reviewer, $F[i]$ can also be obtained by a range quantile query on the wavelet tree; see [11].

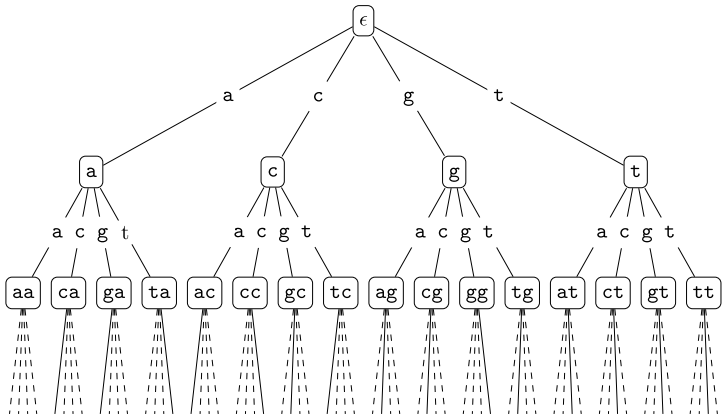

Fig. 3. For these 21 strings there is a procedure call to *getIntervals* if the algorithm gets the string *aacagatccgctggtta*$ as input.

applied if ω has length $|\omega| \geqslant \lceil \log_\sigma n \rceil$. Thus, it must be applied at most $\sum_{i=0}^{\lfloor \log_\sigma n \rfloor} \sigma^i = \frac{\sigma^{\lfloor \log_\sigma n \rfloor + 1} - 1}{\sigma - 1}$ times. Now it follows from $\frac{\sigma^{\lfloor \log_\sigma n \rfloor + 1} - 1}{\sigma - 1} < \frac{\sigma^{\log_\sigma n + 1}}{\sigma - 1} = \frac{n\sigma}{\sigma - 1} \leqslant 2n$ that the procedure *getIntervals* is applied less than $2n$ times. Since every call to this procedure takes $O(\sigma)$ time, the overall worst-case time complexity of the algorithm is $O(n\sigma)$. As a matter of fact, the worst-case occurs when the algorithm gets a de Bruijn sequence as input; see Fig. 3. In combinatorial mathematics, a de Bruijn sequence (named after the Dutch mathematician Nicolaas Govert de Bruijn) of order m on the alphabet Σ' is a cyclic string containing every length m string over Σ' exactly once as a substring. For example, the string *aacagatccgctggtt* is a de Bruijn sequence of order $m = 2$ on the alphabet $\Sigma' = \{a, c, g, t\}$.

7. Conclusions

We presented the first algorithm that computes the longest common prefix array directly on the wavelet tree of the Burrows–Wheeler transformed string (and not on the suffix array). Compared with [1], this article contains several improvements:

- We were able to show that the algorithm runs in $O(n \log \sigma)$ time for a string of length n and an alphabet of size σ.
- Due to a reimplementation of the *getInterval* function, the algorithm is now faster in practice. In addition to that, it now uses some SSE4.2 instructions, so that e.g. the calculation of the leftmost 1-bit in a 64 bit word can be done with an assembler instruction. We believe that our algorithm will profit from further improvements on wavelet trees like the one described in [20].
- We presented an alternative version of our algorithm which needs less space (about $1.5n$ byte) and is only slightly slower.
- It was shown that the new approach can be used in *two* other applications, namely finding shortest unique substrings and shortest absent words.

Our algorithm is available at http://www.uni-ulm.de/in/theo/research/seqana.html.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. Special thanks go to Gonzalo Navarro, whose questions triggered the improved worst-case complexity analysis of Algorithm 2.

References

[1] T. Beller, S. Gog, E. Ohlebusch, T. Schnattinger, Computing the longest common prefix array based on the Burrows–Wheeler transform, in: Proc. 18th International Symposium on String Processing and Information Retrieval, in: Lecture Notes in Computer Science, vol. 7024, Springer-Verlag, Berlin, 2011, pp. 197–208.
[2] N.R. Brisaboa, S. Ladra, G. Navarro, Directly addressable variable-length codes, in: Proc. 16th International Symposium on String Processing and Information Retrieval, in: Lecture Notes in Computer Science, vol. 5721, Springer-Verlag, Berlin, 2009, pp. 122–130.
[3] M. Burrows, D.J. Wheeler, A block-sorting lossless data compression algorithm, Research Report 124, Digital Systems Research Center, 1994.
[4] A. Cox, M. Bauer, G. Rosone, Lightweight BWT construction for very large string collections, in: Proc. 22nd Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 6661, Springer-Verlag, Berlin, 2011, pp. 219–231.
[5] M. Crochemore, F. Mignosi, A. Restivo, Automata and forbidden words, Information Processing Letters 67 (3) (1998) 111–117.
[6] J.S. Culpepper, G. Navarro, S.J. Puglisi, A. Turpin, Top-k ranked document search in general text databases, in: Proc. 18th Annual European Symposium on Algorithms, in: Lecture Notes in Computer Science, vol. 6347, Springer-Verlag, Berlin, 2010, pp. 194–205.
[7] L. Devroye, W. Szpankowski, B. Rais, A note on the height of suffix trees, SIAM Journal on Computing 21 (1) (1992) 48–53.

[8] P. Ferragina, T. Gagie, G. Manzini, Lightweight data indexing and compression in external memory, in: Proc. 9th Latin American Theoretical Informatics Symposium, in: Lecture Notes in Computer Science, vol. 6034, Springer-Verlag, Berlin, 2010, pp. 697–710.
[9] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proc. IEEE Symposium on Foundations of Computer Science, 2000, pp. 390–398.
[10] P. Flick, E. Birney, Sense from sequence reads: Methods for alignment and assembly, Nature Methods 6 (11 Suppl.) (2009) S6–S12.
[11] T. Gagie, S.J. Puglisi, A. Turpin, Range quantile queries: Another virtue of wavelet trees, in: Proc. 16th International Symposium on String Processing and Information Retrieval, in: Lecture Notes in Computer Science, vol. 5721, Springer-Verlag, Berlin, 2009, pp. 1–6.
[12] S. Gog, E. Ohlebusch, Lightweight LCP-array construction in linear time, arXiv:1012.4263, 2011.
[13] R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, in: Proc. 14th Annual ACM–SIAM Symposium on Discrete Algorithms, 2003, pp. 841–850.
[14] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, New York, 1997.
[15] G. Hampikian, T. Andersen, Absent sequences: Nullomers and primes, in: Proc. 12th Pacific Symposium on Biocomputing, 2007, pp. 355–366.
[16] J. Herold, S. Kurtz, R. Giegerich, Efficient computation of absent words in genomic sequences, BMC Bioinformatics 9 (2008) 167.
[17] G. Jacobson, Space-efficient static trees and graphs, in: Proc. 30th Annual Symposium on Foundations of Computer Science, IEEE, 1989, pp. 549–554.
[18] J. Kärkkäinen, Fast BWT in small space by blockwise suffix sorting, Theoretical Computer Science 387 (3) (2007) 249–257.
[19] J. Kärkkäinen, G. Manzini, S.J. Puglisi, Permuted longest-common-prefix array, in: Proc. 20th Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 5577, Springer-Verlag, Berlin, 2009, pp. 181–192.
[20] J. Kärkkäinen, S.J. Puglisi, Fixed block compression boosting in FM-indexes, in: Proc. 18th International Symposium on String Processing and Information Retrieval, in: Lecture Notes in Computer Science, vol. 7024, Springer-Verlag, Berlin, 2011, pp. 174–184.
[21] J. Kärkkäinen, P. Sanders, Simple linear work suffix array construction, in: Proc. 30th International Colloquium on Automata, Languages and Programming, in: Lecture Notes in Computer Science, vol. 2719, Springer-Verlag, Berlin, 2003, pp. 943–955.
[22] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Proc. 12th Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 2089, Springer-Verlag, Berlin, 2001, pp. 181–192.
[23] T.-W. Lam, R. Li, A. Tam, S. Wong, E. Wu, S.-M. Yiu, High throughput short read alignment via bi-directional BWT, in: Proc. International Conference on Bioinformatics and Biomedicine, IEEE Computer Society, 2009, pp. 31–36.
[24] B. Langmead, C. Trapnell, M. Pop, S.L. Salzberg, Ultrafast and memory-efficient alignment of short DNA sequences to the human genome, Genome Biology 10 (3) (2009), Article R25.
[25] M. Léonard, L. Mouchard, M. Salson, On the number of elements to reorder when updating a suffix array, Journal of Discrete Algorithms 11 (2012) 87–99.
[26] H. Li, R. Durbin, Fast and accurate short read alignment with Burrows–Wheeler transform, Bioinformatics 25 (14) (2009) 1754–1760.
[27] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, J. Wang, Soap2: an improved ultrafast tool for short read alignment, Bioinformatics 25 (15) (2009) 1966–1977.
[28] R.A. Lippert, Space-efficient whole genome comparisons with Burrows–Wheeler transforms, Journal of Computational Biology 12 (4) (2005) 407–415.
[29] R.A. Lippert, C.M. Mobarry, B. Walenz, A space-efficient construction of the Burrows–Wheeler transforms for genomic data, Journal of Computational Biology 12 (7) (2005) 943–951.
[30] U. Manber, E.W. Myers, Suffix arrays: A new method for on-line string searches, SIAM Journal on Computing 22 (5) (1993) 935–948.
[31] G. Manzini, Two space saving tricks for linear time LCP array computation, in: Proc. 9th Scandinavian Workshop on Algorithm Theory, in: Lecture Notes in Computer Science, vol. 3111, Springer-Verlag, Berlin, 2004, pp. 372–383.
[32] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Computing Surveys 39 (1) (2007), Article 2.
[33] G. Nong, S. Zhang, W.H. Chan, Linear suffix array construction by almost pure induced-sorting, in: Proc. Data Compression Conference, IEEE Computer Society, 2009, pp. 193–202.
[34] E. Ohlebusch, J. Fischer, S. Gog, CST++, in: Proc. 17th International Symposium on String Processing and Information Retrieval, in: Lecture Notes in Computer Science, vol. 6393, Springer-Verlag, Berlin, 2010, pp. 322–333.
[35] E. Ohlebusch, S. Gog, A. Kügel, Computing matching statistics and maximal exact matches on compressed full-text indexes, in: Proc. 17th International Symposium on String Processing and Information Retrieval, in: Lecture Notes in Computer Science, vol. 6393, Springer-Verlag, Berlin, 2010, pp. 347–358.
[36] D. Okanohara, K. Sadakane, A linear-time Burrows–Wheeler transform using induced sorting, in: Proc. 16th International Symposium on String Processing and Information Retrieval, in: Lecture Notes in Computer Science, vol. 5721, Springer-Verlag, Berlin, 2009, pp. 90–101.
[37] J. Pinho, P. Ferreira, S.P. Garcia, J. Rodrigues, On finding minimal absent words, BMC Bioinformatics 10 (2009) 137.
[38] S.J. Puglisi, W.F. Smyth, A. Turpin, A taxonomy of suffix array construction algorithms, ACM Computing Surveys 39 (2) (2007) 1–31.
[39] S.J. Puglisi, A. Turpin, Space–time tradeoffs for longest-common-prefix array computation, in: Proc. 19th International Symposium on Algorithms and Computation, in: Lecture Notes in Computer Science, vol. 5369, Springer-Verlag, Berlin, 2008, pp. 124–135.
[40] T. Schnattinger, Bidirektionale indexbasierte Suche in Texten, Diploma thesis, University of Ulm, Germany, 2010.
[41] J.T. Simpson, R. Durbin, Efficient construction of an assembly string graph using the FM-index, Bioinformatics 26 (12) (2010) i367–i373.
[42] J. Sirén, Compressed suffix arrays for massive data, in: Proc. 16th International Symposium on String Processing and Information Retrieval, in: Lecture Notes in Computer Science, vol. 5721, Springer-Verlag, Berlin, 2009, pp. 63–74.
[43] J. Sirén, Sampled longest common prefix array, in: Proc. 21st Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 6129, Springer-Verlag, Berlin, 2010, pp. 227–237.
[44] I. Umar, R. Abdullah, Longest-common-prefix computation in Burrows–Wheeler transformed text, in: Proc. 2nd IMT–GT Regional Conference on Mathematics, Statistics and Applications, Universiti Sains Malaysia, Penang, 2006.
[45] Z.-D. Wu, T. Jiang, W.-J. Su, Efficient computation of shortest absent words in a genomic sequence, Information Processing Letters 110 (14–15) (2010) 596–601.